AMPERWARE version 1.0

PRELIMINARY USERS' GUIDE

Scientific Software Products, Inc.

3171 Donald Avenue
Indianapolis, IN 46224
(317) 299-0467

NOTICE

Scientific Software Products, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

Scientific Software Products, Inc. makes no warranties, either express or implied, with respect to this manual or with respect to the software descibed in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Scientific Software Products, Inc. software is sold "as is." The entire risk is as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Scientific Software Products, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Scientific Software Products, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Scientific Software Products, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation may not apply to you.

COPYRIGHT NOTICE

This manual is copyrighted. All rights are reserved except a specifically stated (see below). This document may not, in whole or in part, be copied photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior written consent from Scientific Software Products, Inc. except for Appendix C which may be reproduced and altered in any form for use with software products which utilize **Amperware.**

TRADEMARKS

Apple and Applesoft are trademarks of Apple Computer Co.
DIF is a trademark of Software Arts, Inc.

CONTENTS

Section	Page
INTRODUCTION	1
INSTALLING AMPERWARE	2
VARIABLES	2
SYNTAX	3
METASYMBOLS	4
DEFERRED AND IMMEDIATE MODE	5
INPUT/OUTPUT FACILITIES	7
EDT - input editor	7
PRT - print with upper/lower case	12
OCG - on control code goto	12
DISK COMMANDS	13
SND - send data to disk	13
RCV - receive data from disk	14
FSP - determine free space on disk	14
MISCELLANEOUS	15
JMP - goto a variable	15
GSB - gosub to a variable	15
DAR - delete an array	15
SRT - array sort	15
MCH - substring match	19
COMMAND SUMMARY	A
METASYMBOLS	B
USER EDT TEXT	C

INTRODUCTION

Amperware is a collection of assembly language routines written for the Apple II to enhance Applesoft BASIC. These routines, which are readily accessed from BASIC will add speed and flexibility to your Applesoft programs as well as make coding much more rapid.

Amperware requires that you have Applesoft in ROM, a Language System or any one of the various memory expansion cards into which Applesoft may be loaded. You must also have the Autostart ROM installed since **Amperware** uses some of the firmware routines which it contains. Because of the size of **Amperware** and the space required for the disk operating system (DOS), a minimum of 48 K of RAM is required. This will leave over 33 K for program and variable storage. Finally, a disk drive, while not essential for **Amperware** to function, is needed since many of **Amperware's** features are oriented toward the disk and loading **Amperware** from tape all of the time would be tedious at best. In summary then, you need 48 K, ROM Applesoft (or its equivalent), Autostart ROM and at least one disk drive.

Amperware was developed by Scientific Software Products, Inc. for use in its own software. The package proved so useful that we felt every Applesoft programmer would benefit from it and so offer it as a separate package. A lot of time and hard work went into **Amperware** and we feel it is a good product. So good in fact that we would like to see everyone use it in their commercial products. All we ask before **Amperware** is used commercially is that: (1) we receive a copy of the product and documentation; (2) a copyright notice reading **Amperware is copyright 1982 by Scientific Software Products, Inc.** be displayed prominently in the written documentation and in the program itself and (3) it is understood that all liability for **Amperware's** function lies with you. If you have the courtesy to comply with the above requests, everyone may have a chance to enjoy the benefits of **Amperware.**

If you should decide to use **Amperware** in a commercial product you are welcome to use the text which appears in **Appendix C.** This text is a description of the editing features of the **Amperware** EDT command. You will, of course, need to add a discussion of what various control codes will do since these are dependent on you code.

Before using **Amperware** or running any of the demonstration programs, you should make a back-up copy of the **Amperware** diskette. We have not copy protected **Amperware** in order to make life simpler for you. Please do not violate our trust.

INSTALLING AMPERWARE

There are three steps necessary to fully install **Amperware**. All three steps may be done under program control or in immediate mode. Placing the commands in your program will probably prove to be easier, however. The three steps are:

1. Protect **Amperware** from being overwritten by Applesoft
2. Load the assembly language file AMPERWARE
3. Initialize the ampersand (&) vector

In Applesoft, these steps could be accomplished with the following statements:

```
10 HIMEM=33535
20 PRINT CHR$(4);"BLOAD AMPERWARE"
30 POKE 1013,76: POKE 1014,253: POKE 1015,147
```

You can include these statements at the beginning of each program in which **Amperware** is used or you can include them in your boot disk's HELLO program. If the later option is opted for, you will not have to wait for **Amperware** to load each time you run the program. As long as you do not reset HIMEM, alter MAXFILES, use the FP or INT commands, change the & vector or reboot, **Amperware** will remain in place. You could also destroy **Amperware** by loading a binary file over it so be cautious when you load binary files while **Amperware** is in place.

VARIABLES

Variables used by **Amperware** parameters are just like those used in Applesoft with two extensions. When Applesoft variables are used with these two extensions they become extended variables. The array extension, @, allows an entire array to be referenced without looping or specifying subscripts. This is a particularly useful feature when storing and recalling information from the disk since no FOR ... NEXT loops are required and thus there is no need to keep track of the array size either. If you need a larger array you can change the DIM statement and that's it. Otherwise you would have to search through your Applesoft program to find all of the place in which that size was used in a FOR ... NEXT loop. The second extension, small integer (byte), is indicated by a pounds sign (#). This extension is also most useful in handling data destined for the disk. Positive integers

up to 255 require one byte of memory to represent and those of up to 32767 require two bytes as you probably know. DOS, however, doesn't know much about numbers and stores them as text in its files. That is, for each numeral in your number it uses one byte, plus one for a return character after the numerals. A number like 134 for example would use 4 bytes of disk space to store. If stored as a short integer, however, 134 would require only one byte. A **75%** savings! Not only is disk space save but, as you will soon see, time is saved as well. These extensions will be discussed more fully when the **Amperware** disk commands, **SND** and **RCV**, are presented.

SYNTAX

Syntax is the content and order required in a computer command in order that it will be processed correctly. We will now look at the syntax for the **Amperware** commands in a general way. The specifics of syntax for each command will be presented later.

There are four parts to each **Amperware** command. These are summarized in **Table 1.** The first component, the ampersand (&) is an Applesoft reserved word (page 123 of the Applesoft Manual) which causes an unconditional jump to a specific memory location when it is encountered in an Applesoft program. Advantage is taken of this feature to grab control away from Applesoft and allow **Amperware** to interpret its own commands. The second component of the command is the **mnemonic** which determines which **Amperware** command will be executed. All mnemonics are three letters in length. Note that **letters** as defined in the Applesoft Manual are all upper case. A dash (-) forms the third component of the command and serves to separate the mnemonic from the fourth component which is the parameter list. This is a list of up to eight parameters separated by commas. A few commands, however, violate the parameter construct for reasons of simplicity. In these commands only one parameter is allowed and no letter or equates symbol (=) are required to form the parameter. These exceptions will be discussed as the commands are presented. The reason that parameters were constructed as they were is so that the letter can serve to identify the parameter and thus relieve you from the need to enter parameters in a specific order or to enter some parameters at all.

Table 1 -- Components of Amperware Commands

&	Applesoft reserved word; Indicates an **Amperware** command
mnemonic	three letter code for the command to perform
-	separates the mnemonic and parameter list

parm list mechanism for passing information to the command

Using the metasymbols (see the next section and Appendix B) we can now define the **general** syntax of the **Amperware** commands as follows:

& mnemonic - parameter list

You should note as the commands are presented that some parameters are numeric others are strings and yet others must be variables of one type or another. Further discussion of parameters will be postponed until they are presented in relation to each command. If you just can't wait, Appendix A summarizes the commands, their parameters, the parameter types and whether or not they are required. The spaces in the general syntax are added for clarity and need not be entered for the commands to work. The LIST command in Applesoft will format the commands with the spaces, however, which makes them easier to read.

Commands are not checked for syntactical correctness until the program is executing in either Applesoft or **Amperware.** The Applesoft statements will produce a SYNTAX ERROR IN XXXX message and **Amperware** commands will produce an UNDEF'D STATEMENT ERROR IN XXXX. These two different messages will help you to more easily track down syntax problems.

METASYMBOLS

The Applesoft BASIC Programming Reference Manual (hereafter the Applesoft Manual, Apple Product #A2L0006) uses a series of symbols called metasymbols to help define the syntax for Applesoft commands. We will also use this device. In fact, we will adopt all of the Applesoft Manual's definitions and usages (see pages 29-35 and Appendix N of the Applesoft Manual) and add a few of our own. All of the definitions which are used in this Guide appear in the list below. For further reference see Appendix B in this Guide, which includes a more extensive list of syntactic definitions and abbreviations.

aexpr
:= arithmetic expression
:= avar|real|integer
:= avar subscript
:= (aexpr)
:= [+|-|NOT] aexpr
:= aexpr op aexpr
:= sexpr slop sexpr

Amperware command
:=&mnemonic-parm list

avar
:= arithmetic variable
:= name|name%
:= avar subscript

complex variable
:= var|exvar

comvar
:= complex variable

extended variable
:= #avar|@var|@#avar

exvar
:= extended variable

literal
:= [{character}]

mnemonic
:= a group of three letters which indicate a specific **Amperware** command

parameter
:= a letter followed by an equate symbol (=) and a variable or expression of a specific type determined in part by the letter which precedes it

parameter list
:= a list of up to eight parameters separated by commas

parm list
:= parameter list

sexp
:= string expression
:= svar|string
:= sexpr sop sexpr

svar
:= string variable
:= name$|name$ subscript

DEFERRED AND IMMEDIATE MODE USE OF COMMANDS

As with Applesoft commands, **Amperware** commands may be used in one of two modes: immediate and deferred. In the immediate mode, commands are entered from the keyboard **without** line numbers and are executed immediately. The deferred mode, on the other hand, requires that commands be listed **with** line numbers and are then

executed when the program is RUN. Certain commands do not make sense to use in the immediate mode and should be reserved for deferred mode usage. Other commands are suited for use in either immediate for deferred mode. **Table 2** summarizes the **Amperware** commands and their preferred modes. Any **Amperware** command may be entered in immediate mode but it may not perform correctly so you must think ahead.

Table 2 -- Preferred Command Modes

immediate or deferred	deferred only
EDT	SND
PRT	RCV
FSP	JMP
MCH	GSB
SRT	OCG
	DAR

INPUT/OUTPUT FACILITIES

One of the beauties of the microcomputer is the ability to do outstanding user interaction. This aspect of microcomputing is perhaps the greatest single advantage that the micros have over their larger brothers, the mainframes. **Amperware** allows you to create even more sophisticated user interaction through its EDT, PRT and OCG facilities. (A print using type of function will be added to the package in its final form.) These facilities add features to the user interaction such as inputing commas, colons and qoute marks as well as perform them better and faster than could be done in Applesoft.

EDT -- INPUT EDITOR

EDT - L=aexp[,R=svar,N=avar,D=svar|lit,V=aexp,H=aexp,C=avar,
F=aexp]

EDT is an input editor. Not only does it allow you to gather input from the user through the keyboard but it allows the user to edit his responses as well as some additinal features. Commas, colons and quotes may be entered as well as upper lower case without any hardware modification (Upper case is displayed in inverse video while lower case appears as normal video). Illegal characters are intercepted and disallowed.

Let's begin by discussing how an input collected with EDT looks to the user and then we will discuss just how that is accomplished by the programmer, you. Let's examine some of **Amperware**'s features now. Begin by booting your Apple with the copy you made of the **Amperware** disk. This is necessary in order to install **Amperware** and set the ampersand vecotor as previously described. You should now run the demonstration program EDT DEMO 1 which is on the **Amperware** disk. When you RUN EDT DEMO 1 you will see the following:

PROMPT SOFTWARE__

All of the characters will appear in normal video, which represents lower case. (In this text, lower case characters will be represented with boldface print and upper case with regular density print.) We will get to shifting case in a moment. The cursor is flashing over the S in software. If you would type RETURN at this point the response would be taken as **SOFTWARE**. So what you say. That's just the defintion of a default string. Well, you are correct. Now, however, try copying over to the W by using the right arrow key. What would be response be if you typed RETURN now? No, not **SOFT** but again **SOFTWARE**. This approach of "what you see is what you get" is different from both Applesoft line editing and the INPUT statement but proves to be very useful when you want to edit long lines without copying over

the entire line with the right arrow, a slow process even with the repeat key. While we are on that subject, use the left arrow to move back to the beginning of the line and use the right arrow and repeat keys to copy the entire line. Wow! Pretty fast cursor movement isn't it? Use the left arrow key to move back to the w. But what if you **want** to enter only **SOFT** and **not SOFTWARE** at this point. No problem at all. Just use the CTRL-E key to erase to the end of the line. If you type RETURN now you will get **SOFT** as the response. That's all fine and good but lets get back to **SOFTWARE** now, after all, that's what we are talking about isn't it. Type CTRL-X and your default will be restored no matter how much you have altered the response up to that time. Next, let's try some other editing features. Type CTRL-D and the **S** disappears. This is the delete command. Use CTRL-D three more times to delete **O, F** and **T.** Your screen now looks like this.

```
PROMPT WARE_____
```

Type CTRL-I then ESC and **A.** What, an inverse character? Yes, we have just capatalized a which displays as an inverse character on the Apple's screen (regular typeface in this Guide). The ESC key serves as a shift key. Now type **MPER** and your screen will look like.

```
PROMPT AMPERWARE_
```

With the **A** in inverse video to indicate that it is in upper case. If you type RETURN now the response entered will be AMPERWARE (Amperware). There is also a shift lock mode in which all characters will be entered as upper case. This mode is entered by type ESC twice in succession and is terminated by typing another ESC. One further feature of EDT remains. You may have discovered this feature when you used the repeat key and right arrow to copy over the default string. If not, then try it now. What happens when you reach the end of the line and the cursor is over the last underline. EDT will not let you go beyond that position and beeps at you each time you try. This is the length parameter which EDT controls. Now a user cannot exceed your limits on field lengths and neither do you have to do tedious range checking in your Applesoft code. EDT also has the advantage of giving a visual indication of the maximum length which the field may take on.

How is all this done? Will you have to write tens of lines of code to do it? Is it more trouble than it is worth? Just list the EDT DEMO program and see. What should appear on your screen is:

```
10 REM *** EDT DEMO 1
20 VTAB 10: HTAB 10: PRINT "PROMPT";
30 &EDT - L=12,R=S$,D="software",V=10,H=17
40 GOTO 20
```

Pretty easy isn't it? One **Amperware** command handled all of the input for you and gives you all of the features described above. (The funny characters following the D=" on your screen are the

Apple's way of displaying the lower case letters of software.) The preceding discussion centered on string handling by EDT and ignored numbers for the most part. Fortunately, this was only the discussion's failing and not EDT's. EDT does handle numeric input and does it quite well. Both real and integer inputs are handled with syntax checking at entry. You cannot, for example, enter a period when an integer entry is being made. Let's look at some examples to illustrate these features. If you type RUN EDT DEMO 2 (followed by a RETURN as usual), you see the following on your screen:

INTEGER ____

REAL ____

First, you might try typing some letters (other the E) in response to the **INTEGER** prompt. Nice isn't it? You never have to worry about someone typing their name or place of birth when you want their telephone or social security number. The number formats accept only appropriate characters. Both real and integer types accept digits and plus and minus signs. With a real you may also type a decimal point and the character E for expontentiation. For instance the number 12000 could be entered as 1.2E4. Getting back to our example on the screen, you should now try entering something with a decimal point like 3.14 in the integer field. This illustrates what happens when you try to fool **Amperware**. Once again, **Amperware** saves you a lot of code whiich would otherwise be required to check that inputs which are supposed to be integers are in fact integers. Well, enough about integer inputs, lets get on to **real** business (We can hear the groans at that bad pun already). You should now have a 3 in your integer field from trying to enter 3.14. That's just fine. Press RETURN to move on to the **REAL** prompt and enter 3 for the integer value. Real inputs obviously allow you to enter real numbers including decimal points. Try entering 3.14 once again. This time it works.

Amperware does not prevent you from typing legal characters in an illegal order, however. To do so would have made **Amperware** larger and slower, neither of which we thought you would want. This is not as bad as it might seem though since **Amperware** does ring a bell and prevent you from typing RETURN if you have entered an illegal sequence. Lets look at an example. If you have set up an integer input with a length of three and type **1+1**, **Amperware** will ring the bell each time you type another character including the RETURN key. Therefore, you **cannot** enter a syntactically incorrect numeric expression.

You should also note that, just as when you were entering strings, **Amperware** controlled the length of your response. This length control can be used to limit the range of values which may be entered. For example a three digit integer cannot have a value greater the 999 since there are only three digits. EDT does not do range checking for you in a conventional sense, however. For example it will not prevent a 267 from being

entered if the maximum you want is 254 and you have specified a three digit field length. Also be aware that the field length includes plus and minus signs, decimal points and the E. If, for instance, you want the user to be able to enter decimals numbers up to 10 you must provide three digits (depending on the accuracy that you need) this way the user can enter 10, or 9.9 or 1.6. Concievably, however, a user might enter 9E9 which is clearly greater than ten. That is why you must range check numeric inputs which must be limited.

If you are curious how all of this works and how the numeric inputs differ from string inputs, LIST the program EDT DEMO 2. The program should appear as below:

```
10 REM *** EDT DEMO 2
20 HOME
30 U=0
40 VTAB 10: HTAB 10: PRINT "INTEGER"
50 &EDT - L=5,N=I%,D="",V=10,H=18,F=U
60 VTAB 12: HTAB 10: PRINT "REAL"
70 &EDT - L=5,N=I,D="",V=12,H=18,F=U
80 U=1
90 GOTO 40
```

You should know that EDT will allow illegal numbers to be entered. If you specify a field length of 5 for an integer a user could enter 87694 which is an illegal integer value. EDT does not prevent this value from being entered. When RETURN is typed, however, the system will generate an OVERFLOW error. You may take advantage of this error by useing the ONERROR GOTO facility of Applesoft. If an OVERFLOW error occurs then you can print a message of ring the bell and return to that input with the RESUME statement. In order to illustrate errror trapping of OVERFLOW errors add the following lines to EDT DEMO 2 which should still be in memory.

```
 15 ONERROR GOTO 100
100 PRINT CHR$(7): REM RING HIS BELL
110 RESUME
```

Now, if you RUN the program and try to enter 99999 for the integer value, you will hear the bell and be returned to the input. You will not be able to go on to the REAL prompt until you have entered a valid integer value.

Now, let's look at each of the parameters, what they do and how to use them effectively.

L= -- This parameter determines the maximum length of response that the user can enter. The arithmetic expression is evaluated and the result used as the length. Due to limits placed on Applesoft strings, the maximum length that a response may be is

254. If a longer response is asked for, the number given MOD 256 will be used for the length. You should, therefore, be sure that /aexp/ is less than 255 and greater than 0. (Ever try to enter a string of negative length?) If L=0 is used, then no characters are allowed except control codes, RESET and RETURN. By setting L=0, you can force the user to enter only a control code or RETURN and thus to make a choice of some sort, an option in program control for example.

R= -- The svar specified is the string in which you will find the user's response when EDT returns. If the N parameter is also used, then the the svar will contain the string representation of the number entered.

N= -- The avar specified may be an integer, real or an array reference. If the N parameter is used, then the input collected will be numeric and only numbers, the decimal point, plus and minus signs and the character E for exponentiation will be allowed. Which type of input, integer or real, is collected is determined by the type of variable which N specifies: if it is a real then a real is collected and the decimal and E are allowed, if it is an integer variable type, then only digits the plus and minus signe are allowed. No range checking is done on the number entered and it is possible to collect a number which causes an OVERFLOW error. You can trap for this occurance with the ONERR GOTO statement and handle it merely by RESUMEing the current line, perhaps after printing a warning message or ringing the bell.

D= -- The svar or literal will be used as the default string if it is used. If D is not present then the previous value of the svar specified for R will be used. Failing that the string representation of the avar specified for N will be used. Since either N or R **must** appear in the EDT command, there will always be a default string.

V=,H= -- The V and H parameters are simply vertical and horizontal screen coordinates. They behave just as if you had done an HTAB and VTAB before using the EDT command. They are included in EDT to conserve code space and to make placement of inputs easier. The aexps for V and H are **not** range checked and may produce strange results if 24<V<0 or 39<H<0. These are the same restrictions placed on the HTAB and VTAB commands. This feature is particularly helpful when doing error trapping. On returning to the EDT command, the cursor will be moved back to the correct screen location, no matter what has occured in the interim.

C= -- If a non-editing control code is entered and and the C parameter is used, the ASCII value of the control code will be stored in the avar specified as well as in memory location 254 ($FE). We have not discussed control codes yet and now seems a good time to do so. When the user types a control code during EDT he will hear only a beep unless the C parameter is used. In that case, EDT will return to the Applesoft program with the

value for the control code stored as described. Non-editing control codes may therefore be used for program control or other purposes. (see the OCG command also in reference to control code handling).

F= -- The forms flag, as the F parameter is referred to, is used to chose between displaying the default string and returning only or performing a regular EDT input. The purpose of this flag is to allow you to display an entire screen of fields with their prompts and then allow the user to move through them without duplicating the code using PRINT statements one time and INPUTs the next. To display the default and then return, you should set F=0 by arranging for aexp to evaluate to zero. This may be a complex function or it may be a simple variable assignment. If you run the program FORMS DEMO on the **Amperware** program disk you will see what we mean when we refer to a form. After you have played with this for a while you should list the program and see how it was done. This example can serve as a model for your own programs.

PRT -- PRINT WITH UPPER/LOWER CASE

PRT - svar

The **Amperware** print command, PRT, displays text on the screen using the upper/lower case convention used in EDT. Only strings may be printed and only one string at a time. The text is printed starting at the current cursor position and PRT leaves the cursor at the end of the string just printed. The effect is the same as if you had used the Applesoft PRINT command with a semi-colon after the variable that you printed.

OCG -- ON CONTROL CODE GOTO

OCG - aexp

When using EDT, control codes entered by the user may be trapped and used for program control by using the C parameter. As an additional enhancement **Amperware** provides the OCG command which is On Control-code GOTO. This command will cause control of the the Applesoft program to be transferred to the the line with number aexp if a non-editing control code is entered. This feature is disabled if /aexp/ is 0. If a non-existant line number is specified then, just as in Applesoft, you will get an UNDEF'D STATEMENT ERROR IN linenum.

DISK COMMANDS

Access to the disk has not been overlooked in **Amperware.** One of the most annoying aspects of the Apple is the rather slow disk access. Have you ever tired on waiting for an array of data to be written to the disk? Well, **Amperware** cannot eliminate waits completely but it can make your disk access faster and easier. You will also be able to send and recieve strings which contain characters such a commas, colons and quote marks. The latter is particularly useful which building DIF^{tm} (Data Interchange Format, Software Arts, Inc.) files. These capablilities compliment the EDT features which allow these characters to be entered from the keyboard.

The disk commands use a special form of the parameter list to save code space. For these commands only, the letters and equates symbol are not used. The parameter lists for the disk commands are then simply a list of up to eight complex variables separated by commas. To give this short parmlist a metasymbol defintion:

```
parm list (short form)
          := compvar[{,compvar}]
```

SND - SEND DATA TO DISK

SND - parm list (short form)

The SND command acts much like the Applesoft print command but with some advantages. In order to use SND you must access your file in the usual way using the OPEN and WRITE commands as described in your DOS manual (The DOS Manual, Apple Product #A2L0036 (030-0115-B)). Now you may use the SND statement just as you would PRINT except that: (1) you may now use the extended variable formats (#avar|@#avar|@var) provide by **Amperware,** which allows you to send entire arrays to the disk with only a few characters instead of a FOR ... NEXT loop; (2) you may put up to eight variables on a line with a single SND statement, instead of having to use a separate PRINT statement for each one; (3) any variable types may be interspersed on a line; and, of course, (4) your disk times will be dramatically decreased, thus saving wear and tear on your nerves as well as your disk drives.

There is one restriction which you should know about right away. When you use the extended variable @ to send an array you must somehow keep track of the size of that array for when you use RCV (see below) to read the array back, it must be dimensioned the same size. This requirement can be met in several ways, one of the easiest of which is to simply send the number of dimensions and the dimensions themselves to the file just in front of the array. For example, if you have an array Y$ which is dimensioned

Y$(N,L) you could use the statement **&SND - 2,N,L,@Y$.** While this arrangement may seem confining at first, several new avenues are opened. You can, for instance, concatenate two arrays of the same data type (say reals) mearly be writting them to disk as two separate arrays and reading them back as one. It was this sort of flexibility which prompted us to allow writing and reading of arrays with user defined dimensions instead of storing and using them automatically.

Just how much speed you will gain over PRINT by using SND can be appreciated by studying the graph in **Figure 1** which illustrates the speed at which various data types can be sent to the disk with SND and with PRINT.

RCV - RECIEVE DATA FROM DISK

RCV - parm list (short form)

A RCV command is used to get information from the disk and is to the SND command as INPUT is to PRINT. RCV has the same advantages over INPUT as SND has over PRINT. The speed comparison of INPUT and RCV may be appreciated by examining **Figure 2.**

FSP -- FREE SPACE ON A DISKETTE

FSP - B=avar[,S=aexp,D=aexp]

It is often nice to know the amount of space which remains on a diskette without CATALOGing it and counting sectors. FSP will consult the VTOC (Volume Table of Contents) of the disk in slot S, drive D and determine the number of free blocks and return that value in the avar specified by B. The aexp specified by S and D must evaluate to numbers which are valid slot and drive numbers or the DOS RANGE ERROR will occur. If the S and D parameters are omitted the space on diskette in the last accessed slot and drive will be returned.

MISCELLANEOUS

The following are a collection of miscellaneous routines which we have found to be very helpfull in our Applesoft programming. We think you will find them equally usefull and timesaving.

JMP -- JUMP TO A COMPUTED LINE NUMBER

JMP - aexp

JMP resembles the Applesoft GOTO command except that a variable or other arithmetic expression can be substituted for the line number. If a line with number /aexp/ does not exist then you will get and UNDEF'D STATEMENT ERROR IN XXXX, just as with Applesoft. .cp 10

GSB -- GOSUB TO A COMPUTED LINE NUMBER

GSB - aexp

The GSB facility is to the Applesoft GOSUB statement as JMP is to the GOTO. The aexp is evaluated and treated as a line number. If the line number does not exist then you will get an UNDEF'D STATEMENT ERRROR IN XXX error as in JMP.

DAR -- DELETE A SINGLE ARRAY FROM MEMORY

DAR - @var (var must be the name of an array)

The Applesoft CLEAR statement deletes **all** arrays and variables. At times it is useful to be able to free up the space occupied by a large array which is no longer needed by deleteting it from memory; DAR does just that.

SRT - SORT A STRING ARRAY

SRT - K=var (an array reference),N=aexp[,D=aexp,
I=var% (array reference)]

Sorting an list is a common programming task which is cumbersome and slow if done in Applesoft. SRT allows you to do fast ascending or descending and indexed or non-indexed sorts of any type of Applesoft arrays; string, real or integer. The sort can begin anywhere in the array and extend over any portion of it. SRT is designed primarily for single dimensional arrays but can

be used with multidimensional arrays if some care is exercised. You can select to have the sort into ascending or descending order. For alphanumeric sorts (string arrays), all characters are considered as if they were upper case and all non-numeric, non-alphabetic characters are considered as coming in front of the list; that is, they are considered to be lower than the character A. Finally, you can perform indexed sorts if you chose.

An indexed sort is done by rearranging the values in an integer array which **index** into the array to be sorted rather than rearranging the elements of the array being sorted themseleves. The advantage to this approach is that you can sort the on multiple keys easily. That was quite a bit in once paragraph so lets go back over it slowly and look at an example.

Lets say we have a list of name in array N$ as follows

Jones
Altman
Smith
Pitman
Pitman (this is a different Pitman)

Each of these names in array N$ are associated with a subscript; Jones is N$(1) and Smith is N$(3) for example. If we sort this list is the usual fashion (non-indexed) then it will end up looking like this:

Altman
Jones
Pitman
Pitman
Smith

Now, the names are in order but their subscripts have changed; Jones is now associated with N$(2) and Smith with N$(5). This is alright if this is all we want to do with our list but say that we want to now sort by employee number as well so that the two Pitmans will be placed in order by their employee number but the other names will stay in alphabetical order. You can see where this is leading. Sure it all can be done but only at the expense of a lot of BASIC code and many, many hours of your time trying to catch all of the special situations which are going to occur and let's not even discuss what happens when the program needs to be modified in six months. We must also consider that you may not want you list in alphabetical order for other purposes and who wants to go back to the disk and reread their list of 1000 names or resort the list in memory?

Now let's examine an indexed approach to the problem. Begin with our same list of names but add an integer array I% which contains the subscript numbers of each of the names. Schematically, this looks like:

Jones	1
Altman	2
Smith	3
Pitman	4
Pitman	5

Now, if we sort the list again but this time change only the numbers in the array I% you will see a big difference. After sorting the list we see the following:

Jones	2
Altman	1
Smith	5
Pitman	3
Pitman	4

How is this helpful? Well, in order to access the names in N$ in order you simply loop through the array I% in order and use the values in the array to **index** into the name array, N$. The first name, for example, should be Altman. Using the value in I%(1) to index into the array N$ we get N$(I%(1)) which is Altman, just the name we wanted. Seem like a lot of trouble for nothing? Just consider the circumstance in which you have an entire list of customer names organized by the date that they made a purchase. You want an alphabetical list. Not only will sorting all of those names be slow because of all the string swapping that will have to be done but you list will no longer be in order by date of purchase and you either must resort the list or read it back from the disk. Now do you see the advantage of an indexed sort? In case you still are not convinced you should also note that using the indexed approach prevents you from having to exchange a lot of arrays elements when sorting on a key. In or example of a non-indexed sort, if you had stored the employee number and home addresses of these employees also then you would have to swap these arrays as well as the name array further slowing your sort. Finally, the indexed approach is helpful when you have a multi-key sort to perform. Say you want to sort your employee list by last name and for all those with the same last name you want them sorted by their home address. With an indexed sort you sort them first by home address and then by last name and your task is done. **The second sort uses the order which results from the first sort to begin with.** This feature of SRT can save an immense amount of time and effort.

You should also notice that you can sort a part of an array. This applies to both one and multi-dimensional cases. Say you want to sort only the employees with last names startng with A and you have 14 such employees. Why sort the entire list of employee names? Or consider the case of a mulit-dimensional array in which you want to sort by the first row (the first index in multi-dimensional arrays varies the most rapidly and one full cycle of the earliest index is called a row in this case). If your array were dimensioned 40 by 100 then you could sort the first 40 elements of the array using an indexed sort and then use

the result of that search to index into the entire array. Specifics on sorting parts of arrays will be discussed as the parameters are defined.

K= -- K must specify a valid element in an array. The array may be integer, real or string. If you wish to sort the entire array then the reference would be to subscript 0, if you wished to sort only the last half of an array dimensioned 100 then you would use a subscript of 50: R%(50) for example.

N= -- The value of the aexp specified by N determines the number of array entries which will be sorted by SRT. **You must make certain that the value of the expression is less or equal to the dimension of the array you are sorting.** If you specify a value greater than the dimension then you will **destroy** whatever follows the array in memory. This is not a happy event! If your array is I$ and is dimensiond I$(9,39) then you should not make the value specified by N greater than 400 (9+1 X 39+1). If you do, then you are flirting with disaster. You can also run into trouble with the value of N when working with multi-dimensional arrays. If you want to sort only on the first row of the array and specify an N larger than the dimension plus one of the first row, yo will sort part of the second row along with the first row which is probably not what you want.

D -- Whether the array is sorted in ascending or descending order is determined by the value of D. If D is zero (0), then the array is sorted in ascending order. Any other value of D will cause the array to be sorted in descending order. If D is not specified in a call to SRT it is assumed to be zero.

I -- If you wish to use an indexed sort, you must include an integer array in which the indexes are stored. Before you sort an array for the first time you must initialize the index array. This is easily done with a FOR ... NEXT loop of the form:

```
10 FOR I=1toN
20 I%(I)=I
30 NEXT I
```

If you fail to do this initialization, SRT will try to sort all of the elements you have included which is the zeroeth element only in I% is a newly created array. On the other hand if you have already done an indexed sort on one key and now wish to further sort on another key, you should not initialize I% but rather leave it as is from your previous sort.

If I is not specified, then a regular sort is done and the elements in the array specified by K are actually rearranged.

MCH -- SUBSTRING (SEARCH) MATCH

MCH - S=sexp,M=sexp,P=avar

Applesoft BASIC does not provide a sub-string search function such as the MID$ function found in Microsoft and DEC BASICs. (Note that this MID$ is different from the Applesoft MID$ function which takes a substring out of a string.) Of course your can use the Applesoft MID$ function to find a sub-string but this is extremely slow. The MCH command allows you to rapidly check a string for a sub-string match. Only one string (not string arrays) is checked at a time.

S= -- A string variable specified by S is searched for a match. A literal cannot be used for S.

M= -- The sub-string which is to be found or matched is specified by M. The M reference may be a literal, string variable or string array reference.

P= -- P specifies the arithmetic variable in which the result of the sub-string search is returned. The location of the beginning of the sub-string, in characters, from the beginning of the string referenced by S is returned. If a match is not found a Ø is returned in the avar specified by P.

Appendix A -- Ampersand Command Summary

COMMAND MEMONIC	LETTER	REQ R'D	VARIABLE TYPE	VALUES MIN	MAX
EDT	L	X	aexp	0	254
	R		svar		
	N		avar		
	D		svar|lit		
	V		aexp	0	23
	H		aexp	0	39
	C		avar		
	F		aexp		
PRT	#[1]	X	svar|lit		
OCG	#	X	aexp	0	65535
SND	##		comvar		
RCV	##		comvar		
FSP	B	X	avar		
	D		aexp	1	2
	S		aexp	1	7
JMP	#	X	aexp	0	32767
GSB	#	X	aexp	0	32767
DAR	*[6]	X	@var		
SRT	K	X	var[3]		
	N	X	aexp	0	**[5]
	D		aexp	0	32767
	I		var[4]		
MCH	S	X	svar		
	M	X	sexpr		
	P	X	avar		

(please see notes on the next page)

NOTES:

1 The # symbol implies that a **short** parameter list is used. A short parameter list has one parameter and therefore does not require a letter to designate it.

2 The ## symbol implies a short paramter list is used which may consist of up to eight differnet complex variable separated by commas

3 This must be a reference to an elemnt of a valid array of type real, integer or string

4 This array must be of type integer and you must specify the first element of the array

5 The ** is used to denote the fact that the value specified by N should not exceed the dimensioned value of the array specified by K or I

6 The array specified by K may be integer, real or string but you must use a valid array element name such as T$(1) or Y%(9)

Appendix B -- Metasymbol Dictionary

The Applesoft BASIC Programming Reference Manual (hereafter the Applesoft Manual, Apple Product #A2L0006) uses a series of symbols called metasymbols to help define the syntax for Applesoft commands. We will also use this device. In fact, we will adopt all of the Applesoft Manual's definitions and usages (see pages 29-35 and Appendix N of the Applesoft Manual) and add a few of our own. All of the definitions which are used in this Guide appear in the list below.

```
aexpr
          := arithmetic expression
          := avar|real|integer
          := avar subscript
          := (aexpr)
          := [+|-|NOT] aexpr
          := aexpr op aexpr
          := sexpr slop sexpr

Amperware command
          :=&mnemonic-parm list

avar
          := arithmetic variable
          := name|name%
          := avar subscript


complex variable
          := var|exvar

comvar
          := complex variable

digit
          := 1|2|3|4|5|6|7|8|9|0

extended variable
          :=  #avar|@var|@#avar

exvar
          :=  extended variable

literal
          := [{character}]
```

metasymbol
:= characters used in this document to indicate various structures or relationships in **Amperware** or Applesoft, but which are not part of the package or language themselves
:= |||[|]|{|}|\|~
:= lower-case letter
:= a metaname

mnemonic
:= a group of three letters which indicate a specific **Amperware** command

name
:= real variable name

name%
:= integer variable name

name$
:= string variable name

parameter
:= a letter followed by an equate symbol (=) and a variable or expression of a specific type determined in part by the letter which precedes it

parameter list
:= a list of up to eight parameters separated by commas

parm list
:= parameter list

sexp
:= string expression
:= svar|string
:= sexpr sop sexpr

string
:= "[{character}]"

subscript
:= (aexpr[{,aexpr}])

svar
:= string variable
:= name$|name$ subscript

| | := metasymbol used to separate alternatives

[ ] := metasymbols used to enclose optional material

{ } := metasymbols used to enclose material which may be repeated

\ := metasymbol used to enclose material whose <u>value</u> is to be used: the value of x is written \X\

~ := metasymbol used to indicate a required space